Prequel: A Patch-Like Query Language for Commit History Search

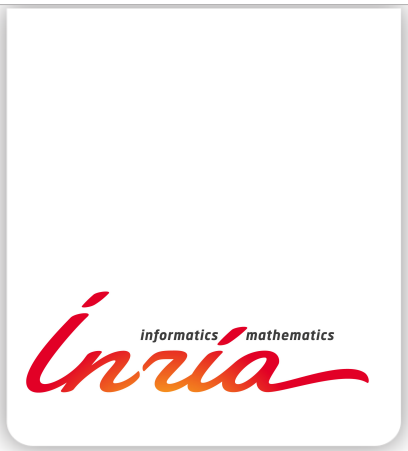
Julia Lawall, Quentin Lambert, Gilles Muller

Prequel: A Patch-Like Query Language for Commit History Search

Julia Lawall, Quentin Lambert, Gilles Muller

RESEARCH REPORT
N° 8918
June 2016
Project-Team Whisper

ISSN 0249-6399 ISRN INRIA/RR--8918--FR+ENG



Prequel: A Patch-Like Query Language for Commit History Search

Julia Lawall*, Quentin Lambert*, Gilles Muller*

Project-Team Whisper

Research Report n° 8918 — June 2016 — 28 pages

Abstract: The commit history of a code base such as the Linux kernel is a gold mine of information on how evolutions should be made, how bugs should be fixed, etc. Nevertheless, the high volume of commits available and the rudimentary filtering tools provided mean that it is often necessary to wade through a lot of irrelevant information before finding example commits that can help with a specific software development problem. To address this issue, we propose Prequel (Patch Query Language), which brings the descriptive power of code matching to the problem of querying a commit history. We show in particular how Prequel can be used in understanding how to eliminate uses of deprecated functions.

Key-words: Linux kernel, code search, software mining

* Sorbonne Universités/UPMC/Inria/LIP6

RESEARCH CENTRE
PARIS – ROCQUENCOURT

Domaine de Voluceau, - Rocquencourt
B.P. 105 - 78153 Le Chesnay Cedex

Prequel: Un langage à la patch pour l'interrogation des bases de commit

Résumé : L'histoire des commits dans une base de code comme le noyau Linux est une mine d'or d'informations décrivant comment les évolutions doivent être faites, comment les bugs doivent être corrigés, etc. En revanche, le grand volume de commits disponibles et la disponibilité d'outils de filtrage rudimentaires impliquent qu'il est nécessaire de dépouiller de nombreuses informations irrelevantes avant de trouver les exemples qui peuvent aider à résoudre un problème spécifique de développement logiciel. Dans ce rapport, nous proposons le langage Prequel (Patch Query Language), qui offre la puissance descriptive de la reconnaissance de code au problème de l'interrogation d'une base de commit. Nous montrons en particulier que Prequel peut être utilisé pour éliminer et remplacer les utilisations de fonctions dépréciées.

Mots-clés : Noyau Linux, Reconnaissance de code, fouille de dépôts logiciels

1 Introduction

Large infrastructure software, such as the Linux kernel, must continually evolve, to meet new requirements on performance, security, and maintainability. Such evolution means that it is essential for developers to keep up to date with the latest APIs and their usage protocols. A number of recent approaches address these issues by scanning the current state of the code base for examples of API usages and using statistical methods to highlight the most common patterns [2, 8, 9]. These approaches, however, miss a key element of understanding the current state of the code: its history, *i.e.* how did the code come to have its current form, and what are the possible alternatives. Indeed, this kind of information is essential for a developer who must modernize an out of date code base, or who must backport a new functionality to an older version [15].

Today, history information is freely available for most large open-source infrastructure software projects, via a version control system, such as git. A version control system records the set of changes that have been committed to the code base over time, with each commit being accompanied by a log message describing the reasons for the change. The Linux kernel in particular furthermore follows the strategy that each commit should involve only one logical change [7], suggesting that it should be possible to find commits that provide specific information about why a change is needed and how it should be carried out.

Despite the potential value of the information stored in the commit history, this information is difficult to extract in practice, mainly due to the amount of information available. For example, between the recent versions Linux 4.3, released in November 2015, and Linux 4.4, released in January 2016, which represents only a small portion of the 22-year development history of the Linux kernel, there were 14,082 commits, comprising 750588 added lines of code and 507175 removed lines of code. Furthermore, the granularity of these commits varies widely, depending on the purpose of the change. For example, a commit adding or removing a large function definition may involve hundreds of lines of code, and thus is not likely to be useful in understanding the motivation behind a specific code fragment found within this function definition. Git filtering commands `git log -G` and `git log -S` restrict the set of commits displayed to those in which at least one changed line matches a particular regular expression, but provide no control over the granularity of the commits that are returned. Furthermore, there is no way to provide multiple keywords, to describe properties of the lines of code that are not changed, or to specify any semantic relationships between the changed lines. As a result, these commands may return many irrelevant results, that the developer has to manually search through to find the one that contains the information that is most relevant to a given development problem.

To address the difficulty of finding information in a C code history, we take inspiration from previous work on specifying transformations for C code. The tool Coccinelle [1, 12] and its associated Semantic Patch Language (SmPL) propose a transformation specification notation that is based on the familiar patch syntax. We observe that such a specification of how to transform code can also be viewed as a description of the effect of the transformation process. That is, a specification of which lines to add and remove can also be viewed as a description of the lines that have been added and removed, after the transformation has been performed. In this paper, we thus explore the usability of the SmPL notation as a patch query language, providing a description, which we refer to as a *patch query*, of the effect of a previous transformation process, and an associated tool, Prequel, for applying this description to the patches found in a series of commits.

To support the use of a SmPL-like language as a patch query language, we must address the following issues:

- Expressivity: When searching through commits, we typically do not know, *a priori*, the complete set of changes that are performed. What features should a patch query language

offer to allow the user to query a code history in an approximate way?

- Relevance of the results: Given that a patch query is intended to provide only an approximate description of a change, how can Prequel provide the user with only the most relevant results?
- Performance: The rate of change in the Linux kernel implies that finding commits that exhibit a particular change may require examining hundreds of thousands of commits. How can we provide a level of performance that is acceptable for interactive use?

The contributions of this paper are as follows:

- We present the Patch Query Language (PQL), its implementation and the Prequel run-time infrastructure.
- We evaluate Prequel on some typical Linux kernel development tasks, in terms of expressiveness and performance, and by comparison to existing tools.
- We show that the performance of Prequel is sufficient to process all of the 280,901 commits between Linux v3.0 and v4.4 (4.5 years) in a reasonable amount of time.

The rest of this paper is organized as follows. Section 2 provides some background for our work, including properties of the evolution of the Linux kernel and an overview of Coccinelle. As a motivating example, Section 3 presents a case study involving an API evolution task and the assistance that is provided by git. Section 4 revisits and extends this case study using Prequel. Section 5 presents the Prequel language and its implementation by compilation into SmPL. Section 6 presents the Prequel run-time system. Section 7 evaluates the expressiveness and performance of Prequel. Finally, Section 8 describes related work, and Section 9 presents conclusions and directions for future work.

2 Background

In this section, we present the starting points of our work: the Linux kernel, the notion of a patch, the version control system `git`, and the SmPL language.

2.1 Linux kernel

The Linux kernel is an open source operating system kernel that has been in active development since 1994. As of Linux v4.4, released in January 2016, which we use as the reference version in this paper, the Linux kernel amounts to over 13.5 million lines of C code. The code base is decomposed into various subsystems, for device drivers, network device drivers, memory management, etc. Each subsystem defines a collection of in-kernel API functions to be used at that level and in more specific subsystems. This results in a very large collection of API functions with more or less specific purposes, and that are familiar to a more or less restricted set of developers. Finally, the Linux kernel has a developer base with very diverse levels of knowledge about the code, with a core that has worked on the kernel for many years, and others who have contributed, *e.g.*, a single device driver, or a single patch.

The Linux kernel adopts a strategy of allowing the in-kernel APIs to evolve freely, without the constraint of maintaining backwards compatibility. This strategy makes it possible to deprecate functions, and potentially to replace them with new ones, to optimally address performance, security, and maintainability concerns. This strategy, however, also raises challenges for developers and maintainers who then need access to accurate information on how to modernize their code.

```
1 diff --git a/drivers/scsi/isci/init.c b/drivers/scsi/isci/init.c
2 index 695b34e..4198e45 100644
3 --- a/drivers/scsi/isci/init.c
4 +++ b/drivers/scsi/isci/init.c
5 @@ -356,7 +356,7 @@ static int isci_setup_interrupts(struct pci_dev *pdev)
6   for (i = 0; i < num_msix; i++)
7     pci_info->msix_entries[i].entry = i;
8
9 - err = pci_enable_msix(pdev, pci_info->msix_entries, num_msix);
10 + err = pci_enable_msix_exact(pdev, pci_info->msix_entries, num_msix);
11   if (err)
12     goto intx;
```

Figure 1: Patch for Linux kernel commit e85525c

2.2 Patches

A patch is a line-based description of a set of code changes [10]. A patch can be generated by applying the tool `diff -u` to the code as it exists before and after the change. Patches are also commonly used by version control systems for describing the code history.

Figure 1 shows an example of a patch, obtained from Linux kernel commit e85525c. For each change made by the commit, the patch indicates the affected file, and then describes the change in a line-based manner. Specifically, a change is described in terms of affected line number information (*e.g.*, line 5), followed by some context (unchanged) lines, which are unannotated (*e.g.*, lines 6-8 and 11-12), some removed lines, which are annotated with `-` in the first column (*e.g.*, line 9), and some added lines, which are annotated with `+` in the first column (*e.g.*, line 10). A sequence of added and removed lines, with no intervening context lines, is referred to as a *hunk*. A patch may describe changes in multiple files, and the changes for each file may consist of multiple hunks.

2.3 Git

Git[1] is a distributed version control system that was originally developed for the Linux kernel and that was first used with Linux v2.6.12, released in June 2005. It is now used for many open source projects. Git provides various commands for visualizing patches. Particularly relevant to our work are `git show`, which shows the contents of the patch associated with a given commit, and `git log -G`, which searches for commits that contain a line that adds or removes code matching a given regular expression.[2]

2.4 SmPL

A patch is a simple and precise specification of a transformation, but it can only be applied to code that is at least very similar to the code from which it was generated. The Semantic Patch Language (SmPL),[3] supported by the Coccinelle program transformation tool, generalizes the patch notation with *metavariables*, which can match arbitrary terms, and with the path operator "`...`", which matches an arbitrary code sequence.

A simple SmPL semantic patch is shown in Figure 2. This semantic patch consists of a single rule, which is divided into two sections: declaration of metavariables between the initial pair

[1] https://git-scm.com/

[2] Git provides a related command `git log -S` that selects commits that change the number of occurrences of a given string in the affected code. We do not use this command, because of the possibility of false negatives, if the string is removed at one point in the code, and then added elsewhere in an unrelated way.

[3] http://coccinelle.lip6.fr/docs/

```
@@
expression x;
@@
  a();
  ... when != x
- b(x,x);
+ c(x);
```

Figure 2: SmPL example

of `@@` delimiters, followed by a pattern describing how the transformation should be performed. This semantic patch rule declares a single metavariable `x` that can match any expression. The pattern then indicates that a call to the function `a`, with no arguments, that is followed along all possible intraprocedural control-flow paths, as indicated "`...`", by a call to `b` with two identical arguments, should be modified by removing the call to `b` and replacing it with a call to `c` with `x` as its single argument. The annotation `when != x` on the "`...`" furthermore indicates that there should be no reference to `x` between the call to `a` and the call to `b`. In the more general case, a semantic patch can consist of multiple rules, a rule can be declared to depend on the success or failure of the matching of other rules, and information can be passed between rules via inherited metavariables. Semantic patches can also include scripts written in Python or OCaml for further processing of the matched code fragments. Scripts likewise consist of metavariable declarations, which are typically inherited from pattern matching rules, followed by arbitrary code in the scripting language.

3 Case Study

In this section, we present a case study that we then use as a running example in the rest of the paper. Our case study centers on the problem of enabling Message Signaled Interrupts (MSI) in the Linux kernel, typically in network drivers. We have deliberately chosen a functionality that is somewhat specialized, to represent the case of an API whose usage is not common knowledge, and where access to history information may thus be helpful. Furthermore, this example illustrates the case where modernizing the use of a deprecated function requires making some choices, and the motivations for those choices disappear in the resulting code.

3.1 Background

Message Signaled Interrupts enable a PCI device to signal interrupts by writing data to a memory address. Prior to 2014, device drivers enabled Message Signaled Interrupts using the function `pci_enable_msix`. This function takes as an argument the number of interrupts to enable. If only a smaller number of interrupts can be enabled, the return value is the number of available interrupts, and the device driver can try again with the returned value.

In 2014, the function `pci_enable_msix` was deprecated for use by device drivers, and was replaced by the function `pci_enable_msix_range` in a patch dated January 13, 2014, and by a specialized variant `pci_enable_msix_exact` on February 13, 2014. `Pci_enable_msix_range` is to be used when the driver can tolerate a smaller number of enabled interrupts, within a range. It returns the number of enabled interrupts on success. The function `pci_enable_msix_exact` is to be used when the driver requires that all of the requested Message Signaled Interrupts be enabled. It returns 0 on success. `Pci_enable_msix_exact` is simpler to use and maintain, in that it takes fewer arguments and has a success return value, 0, that is more common in kernel code.

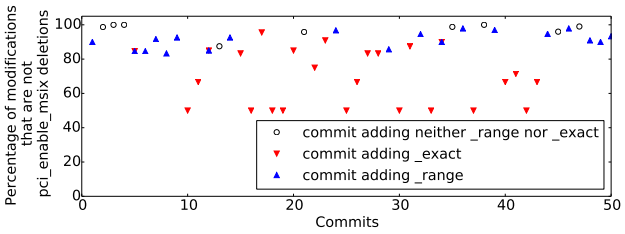


Figure 3: Percentage of modified lines other than those that remove `pci_enable_msix`. In this case, the lower bound is always 50%, which occurs in the case of replacement of `pci_enable_msix` by another function call.

Even though `pci_enable_msix` is deprecated for use in drivers, there remain four calls to `pci_enable_msix` outside of the PCI MSI library in the Linux kernel v4.4 code, released in January 2016. All occur in code that was introduced after February 2014, showing that developers are not always aware of the latest API functions.

3.2 Analysis using git

We put ourselves in the position of a developer who knows that the function `pci_enable_msix` is deprecated, but does not know what are its possible replacements. Such a developer would thus like to find examples of commits that replace a call to `pci_enable_msix` with a call to some other function. We consider how this can be done using git.

One option is the command `git log -G pci_enable_msix`, which finds the commits, in reverse chronological order, that contain an added or removed line that mentions `pci_enable_msix`. Figure 3 illustrates the properties of the patches returned by this command. In this graph, each point on the x-axis represents a commit, with the leftmost point being the most recent one, as the most recent one is produced first by `git`. The y-axis represents the percentage of changed lines that do not explicitly remove a call to the function `pci_enable_msix`. Commits that replace a call to `pci_enable_msix` by a call to `pci_enable_msix_exact` are highlighted by a red downward triangle and commits that replace a call to `pci_enable_msix` by a call to `pci_enable_msix_range` are highlighted by a blue upward triangle. Hollow circles represent commits that affect calls to `pci_enable_msix` in some other way. While the first commit does replace `pci_enable_msix` by `pci_enable_msix_range`, the next three are irrelevant to the evolution problem. The developer then has to take the time to go four commits further in the history to discover that there is an alternate option, provided by the function `pci_enable_msix_exact`. Furthermore, we see that the percentage of changed lines that do not explicitly remove a call to the function `pci_enable_msix` is high in the most recent patches, suggesting that the change is complex and thus fully understanding the options requires studying a good number of patches. Figure 4 shows the cumulative number of lines of changes before reaching each of the patches that replaces a call to `pci_enable_msix` by either `pci_enable_msix_range` or `pci_enable_msix_exact`. This figure shows that it is necessary to scroll through many changed lines when using `git log -G` to visualize all relevant commits.

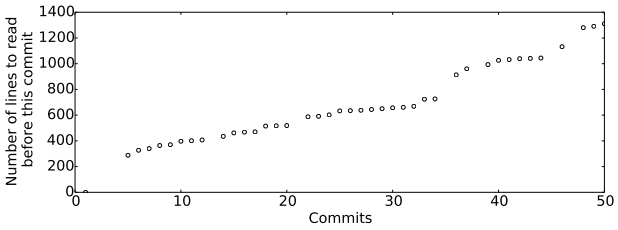


Figure 4: Number of changed lines in patches reached more recent than those replacing `pci_enable_msix` by either `pci_enable_msix_exact` or `pci_enable_msix_range`

4 Using Prequel

We now consider how we can use Prequel to understand how to enable Message Signaled Interrupts.

4.1 Searching for replacements of `pci_enable_msix`

In our first experiment with Prequel, we again put ourselves in the position of a developer who knows that the function `pci_enable_msix` is deprecated, but does not know its possible replacements.

The Prequel patch query for searching for commits that illustrate the modernization of calls to `pci_enable_msix` is shown in Figure 5. This patch query consists of a pattern-matching rule `r` (lines 1-9) followed by a Python script (lines 11-14). The rule `r` declares three metavariables (lines 2-4): an expression `x`, to represent the return value of the call, an expression list `es`, to represent the arguments of the call, and an identifier `f` that is different than `pci_enable_msix` to represent the name of the function by which `pci_enable_msix` is replaced. The pattern then has the form of a function call (lines 6-9), where the function name `pci_enable_msix` is replaced by the name of an arbitrary function `f`. The Python script inherits (line 12) and then prints the name of the replacement function `f` (line 14). The syntax of this patch query is similar to that of the SmPL semantic patch shown in Figure 2, but the semantics is different, in that the `-` and `+` annotations on lines 7 and 8 do not represent changes that should be performed on a source code file, but rather tokens that should be matched on the `-` and `+` lines of a patch. Given this patch query, Prequel prints the identifiers of the matching commits that meet the criteria specified by the user for the percentage of lines and hunks matched, and for each such commit prints the text generated by the Python code.

The pattern-matching rule `r` contains a mixture of context code, *i.e.*, the assignment of `x` and the argument list `es`, and modified code, *i.e.*, the name of the called function. As the Prequel user may not fully understand the code history, an underlying assumption of Prequel is that the user may not know about all of the changes that have taken place in the code. Accordingly, while the code annotated with `-` or `+` in the patch query *must* appear on the changed lines in any matched patch, the unannotated code *can* appear on such lines as well. Annotations on metavariables indicate how much flexibility is allowed in the matching process. The metavariable `x` has no annotation, and thus it is said to be *fixed*, meaning that it must match the same expression before and after the patch application. On the other hand, the metavariable `es` is annotated as `flexible`, meaning that it can match one sequence of arguments before the patch application, and a different sequence of arguments afterward. Indeed, `pci_enable_msix` takes three arguments, while one possible result of modernizing `pci_enable_msix`, *i.e.*, `pci_enable_-`

```
@r@
expression x;
flexible expression list es;
identifier f != pci_enable_msix;
@@
x =
- pci_enable_msix
+ f
  (es)

@script:python@
f << r.f;
@@
print f
```

Figure 5: Patch query searching for replacements of `pci_enable_msix`

`msix_range`, takes four, also including the start of the range.

A design choice of this patch query is the use of a pattern having the form of an assignment of some fixed expression `x` to the result of the call `pci_enable_msix`. This design choice restricts the set of results that can be obtained, because it only matches calls that are used in assignments, and then only in assignments where the left-hand side does not change. The use of an assignment to the fixed `x`, however, does help ensure that there is some relationship between the call to `pci_enable_msix` before the patch application and the call to the unknown function `f` after the patch application. Otherwise, Prequel would match `f` to any called function in the same hunk. Other kinds of context information could be used to connect the calls to `pci_enable_msix` and `f`, such as requiring that the left hand side of the assignment have the same type, but not necessarily be the same code, or requiring that the two calls have some argument in common. The user can explore such alternate patterns if the set of results obtained with a given pattern is not sufficient.

The patch query shown in Figure 5 when applied to the complete Linux kernel history between v3.0 and v4.4 produces information about 46 commits. Extracts of the results are shown in Figure 6. The output consists of a series of records, containing the commit identifier, the percentage of matched lines, and any output generated by the patch query's script code. From this output, one can see immediately that the possible replacements for `pci_enable_msix` are `pci_enable_msix_range` or `pci_enable_msix_exact`. Furthermore, the higher percentages associated with the commits that produce calls to `pci_enable_msix_exact`, suggest that the transformation to use `pci_enable_msix_exact` typically requires fewer additional changes than the transformation to use `pci_enable_msix_range`; this may reflect how much difficulty will be required to understand each of the transformations and the tradeoff between them. Unlike git, which simply scrolls the commit logs and patches linearly from the most recent backward in time, the information provided by Prequel gives the developer a concise overview of the relevant commits and can help orient the developer in his next steps to understand the evolution process. We elaborate on this use of Prequel in the next section with a more precise study of the conversion of `pci_enable_msix` to `pci_enable_msix_range`.

Figure 7 shows an extract of the patch associated with commit 4f871e1, dated September 3, 2014, which has a match rate of 10% and uses both of the new functions in different contexts.

4.2 Further investigation of the `pci_enable_msix_range` case

To refine the results obtained using the previous experiment, we focus on how to use Prequel to study the question of when to use the function `pci_enable_msix_range` based on the context in which calls to `pci_enable_msix` occur. Note that this study does not have a counterpart with

```
471212d: 100%                   ...
pci_enable_msix_exact           bf3f043: 6%
efdfa3e: 100%                   pci_enable_msix_range
pci_enable_msix_exact           a9df862: 6%
e85525c: 100%                   pci_enable_msix_range
...
4f871e1: 10%
pci_enable_msix_exact
pci_enable_msix_range
```

Figure 6: Output of the patch query shown in Figure 5

```
1 diff --git a/drivers/scsi/lpfc/lpfc_init.c b/drivers/scsi/lpfc/lpfc_init.c
2 index 990c3a2..1953b3b 100644
3 --- a/drivers/scsi/lpfc/lpfc_init.c
4 +++ b/drivers/scsi/lpfc/lpfc_init.c
5 @@ -8232,2 +8232,2 @@ lpfc_sli_enable_msix(struct lpfc_hba *
6 - rc = pci_enable_msix(phba->pcidev, phba->msix_entries,
7 -                      ARRAY_SIZE(phba->msix_entries));
8 + rc = pci_enable_msix_exact(phba->pcidev, phba->msix_entries,
9 +                           LPFC_MSIX_VECTORS);
10 @@ -8800,7 +8798,3 @@ lpfc_sli4_enable_msix(struct lpfc_hba *
11 -enable_msix_vectors:
12 - rc = pci_enable_msix(phba->pcidev,
13 -                      phba->sli4_hba.msix_entries, vectors);
14 - if (rc > 1) {
15 -    vectors = rc;
16 -    goto enable_msix_vectors;
17 - } else if (rc) {
18 + rc = pci_enable_msix_range(phba->pcidev,
19 +                           phba->sli4_hba.msix_entries, 2, vectors);
20 + if (rc < 0) {
```

Figure 7: Patch illustrating uses of `pci_enable_msix_exact` and `pci_enable_msix_range`

git alone (Section 3), because git only searches for a single token, independent of the context in which it occurs.

Given the results (Figure 6) obtained using Prequel with our original patch query, a natural starting point is to study the commits that produce calls to `pci_enable_msix_range` and that have the highest match rate. Three such commits have a match rate of 40%. In all three cases, however, the start and end arguments of the range (the third and fourth arguments of `pci_enable_msix_range`) are the same, and thus it seems that `pci_enable_msix_exact` should have been used. These commits are thus not helpful in determining how to use `pci_enable_msix_range` correctly. We thus turn to the commits that have a lower match rate. The match rates range from 33% down to 6%, suggesting that there is a lot of variability in how the transformation is carried out.

We then may study a few patches, such as the one in Figure 7, combined with the definitions of `pci_enable_msix` and `pci_enable_msix_range`, to find some common patterns. The results of this study suggest that a common pattern, as illustrated in Figure 7, is to call `pci_enable_msix` (lines 12-13), then possibly store the return value, representing the number of interrupts that are actually available, in the variable that is used as the third argument of the call to `pci_enable_msix` (line 15), and then call `pci_enable_msix` with this updated value (lines 12-13). In Figure 7, these steps are done in a loop, implemented using a `goto` and a label, but other patches identified by Prequel show code using a `while` loop, or simply a sequence of calls. Furthermore, in the latter case, the return value may be used directly as the third argument, rather than reusing the original one.

While the above description characterizes a number of patches, we also need to know whether

```
1 @double_enable1 exists@
2 expression e,e1,e2,e3,e4;
3 position p;
4 @@
5   e =
6 - pci_enable_msix@p(e1, e2, e4)
7 + pci_enable_msix_range(e1, e2, e3, e4)
8   ...
9 - e4 = e
10  ...
11 - pci_enable_msix(e1, e2, e4)
12
13 @double_enable2 exists@
14 expression e,e1,e2,e3,e4;
15 position p;
16 @@
17   e =
18 - pci_enable_msix@p(e1, e2, e4)
19 + pci_enable_msix_range(e1, e2, e3, e4)
20  ...
21 - pci_enable_msix(e1, e2, e)
22
23 @should_be_exact@
24 expression e1,e2,e3;
25 position q;
26 @@
27 + pci_enable_msix_range@q(e1, e2, e3, e3)
28
29 @unanticipated@
30 expression e,e1,e2,e3,e4;
31 position p != {double_enable1.p,double_enable2.p};
32 position q != should_be_exact.q;
33 @@
34   e =
35 - pci_enable_msix@p(e1, e2, e4)
36 + pci_enable_msix_range@q(e1, e2, e3, e4)
```

Figure 8: Patch query detecting unanticipated uses of `pci_enable_msix_range`

there are patches that it does not characterize, *i.e.*, whether there are conditions that our study has not taken into account. To perform an exhaustive search, we again turn to Prequel. Figure 8 shows a patch query that detects the above cases: a double `pci_enable_msix` call with reassignment (`double_enable1`, lines 1-11), a double `pci_enable_msix` call using the return value (`double_enable2`, lines 13-21), and a call to `pci_enable_msix_range` that should use `pci_enable_msix_exact` (`should_be_exact`, lines 23-27). The patch query then reports on replacements of `pci_enable_msix` by `pci_enable_msix_range` that do not satisfy any of these patterns (`unanticipated`, lines 29-36).[4] Only the matches of `unanticipated` are reported, because the other rules are only used in negative dependencies (lines 31 and 32).

This refined patch query matches 6 commits, with match rates of 13%–50%. In three cases, the update to the third argument of `pci_enable_msix` is retained after the commit, rather than being removed as required by the rule `double_enable1`. In 3 cases, including one of the previous cases, the third argument for the second call to `pci_enable_msix` is computed in some other way than simply taking the return value of the first call. Finally, in one case, the value returned

[4]This patch query uses some features borrowed from SmPL that have not previously been presented. The `exists` annotation (lines 1 and 13) indicates that the pattern matches if there exists a control-flow path that satisfies the pattern; the default is that all control-flow paths starting from term matching the beginning of the pattern must satisfy the complete pattern. A `position` metavariable (lines 3, 15, and 31-32) collects information about the position of the match of the token to which the position metavariable is attached. Finally, a metavariable can be *inherited* from earlier rules (lines 31-32), by mentioning the name of the rule in which it is defined in the current rule's list of metavariable declarations, and a position variable can be declared to match a position different than the positions matched by one or more inherited position variables (lines 31-32).

by `pci_enable_msix` is in turn returned by the enclosing function, so that its caller can decide how to proceed. This variety of issues has to be taken into account by anyone wanting to apply the evolution to their code.

5 Compilation of PQL into SmPL

Prequel is implemented by compiling PQL code into SmPL, and using Coccinelle to match the resulting SmPL code against snapshots of the files affected by a commit before and after the application of the commit's patch. We begin by describing the differences between PQL and SmPL, many of which are motivated by the patch query compilation process. We then describe the compilation process itself, which involves four steps: 1) marking the parts of the patch query that should match changed code, 2) classifying rules according to whether they should match code before the application of the commit's patch, after the application of the commit's patch, or both, 3) separating the patch query into splices that should be matched against the before and after code, respectively, and 4) inserting script code to synchronize the results.

5.1 Prequel Patch Query Language

PQL follows the same design as SmPL: specifications amount to fragments of C code, decorated with `-` and `+` annotations. In addition to fragments of C code, PQL and SmPL provide a few operators for describing control-flow paths, of which we have seen the example of "`...`" and "`when`" in Figure 2. The differences between PQL and SmPL are as follows:

Symmetric use of `-`, `+`, and positions: PQL, as it has the goal of matching against existing patches, allows the `-` and `+` annotations and position variables to be used on any token in a patch query. SmPL, on the other hand, uses a semantic patch as a directive on how to perform a transformation, and thus allows `-` to be used freely, but requires `+` to be used on tokens that are near `-` or context code, representing the source code of the transformation. SmPL furthermore only allows positions on `-` and context code. These differences are illustrated by the rule `should_be_exact` in Figure 8, which is allowed in PQL but not in SmPL.

`flexible` metavariables: As illustrated by the declaration of the metavariable `es` in the rule `r` in Figure 5, PQL allows metavariables to be declared as `flexible`, indicating that they need not match the same code fragment in the before and after code.

`when-` and `when+`: PQL introduces `when-` and `when+`, which put separate constraints on the control-flow path in the before and after code, respectively (*cf.* Figure 2).

Constraints on disjunctions: SmPL allows expressing a disjunction of transformations, such as `( - a + c | - b + c )` (written inline for conciseness), indicating that either `a` should be replaced by `c` or `b` should be replaced by `c`. Because Prequel matches the patch query code annotated `-` independently from the patch query code annotated `+`, it is not able to decide between the two instances of `+ c` in the above disjunction. Prequel thus disallows disjunctions that mix `-` and `+` code, as well as disjunctions that mix `-` code and context code, and that mix `+` code and context code. Typically, the user can reorganize the patch query to avoid these restrictions.

Constraints on compiler directives and fresh identifiers: Coccinelle supports rules that add compiler directives, but does not support matching against such directives in the source code. Likewise, Coccinelle support the creation of fresh identifiers for use in added code, but does not support the detection of the freshness of an identifier. As Prequel only uses the matching functionality of Coccinelle, it does not support either of these features.

5.2 Idealized syntax

For the purpose of formalizing the Prequel compiler, we consider the following simplified patch query language:

$$
\begin{array}{rcl}
A & ::= & \texttt{-} \mid \texttt{+} \mid \texttt{-+}t^+ \mid \varepsilon \\
t & ::= & \mathsf{id} \mid \mathsf{metaid} \mid \texttt{(} \mid \texttt{)} \\
P & ::= & exp^A\ \texttt{(}^A\ exp^A\ \texttt{)}^A \\
exp & ::= & \mathsf{id} \mid \mathsf{metaid} \\
M & ::= & \texttt{fixed}\ \mathsf{metaid}\texttt{;}M \mid \texttt{flexible}\ \mathsf{metaid}\texttt{;}M \mid \varepsilon \\
Dba & ::= & \texttt{before} \mid \texttt{after} \mid \texttt{before after} \mid \varepsilon \\
Dr & ::= & \mathsf{rule\_name}\ Dr \mid \varepsilon \\
R & ::= & \mathsf{rule\_name} \times Dba \times Dr \times M \times P \\
SP & ::= & R\ \overline{SP} \mid \varepsilon
\end{array}
$$

In this language, a patch query, SP, is composed of a sequence of rules R. Each rule has a name, may be specified to match against before code, after code, or both, or be left unspecified, and may be specified to depend on the successful matching of all of a set of other rules (Dr). A rule may declare some metavariables, which may be declared as *fixed* or *flexible* (M). A fixed metavariable must have the same value when matching both the before and after code, while a flexible metavariable can have different values in these two cases. The body of a rule is a pattern (P) that, in our simplified language, matches a single function call of one argument, $exp^A\texttt{(}^A\ exp^A\ \texttt{)}^A$. The expressions in the function and argument position can be either explicit names or metavariables (exp). This pattern language is very much simplified, for illustration; indeed, the pattern code is mostly orthogonal to the compilation process, and is mainly simply propagated to Coccinelle. A pattern P can also be viewed as being made up of a sequence of tokens, t. Each token in a pattern may be annotated with a transformation A that indicates whether the token is removed from the before code (`-`), added to the after code (`+`), or a single token is removed and is replaced by a sequence of tokens (`-+`).

Observe that this language allows nonsense patterns, such as $\texttt{a(}^{\texttt{-+x)(}^{\texttt{y}}}\texttt{b)}$, in which the combination of the code annotated with `+` and the context code, *i.e.*, `a x)(y b)`, does not match any valid C term. To be able to eliminate nonsense patterns, we define the notion of a slice on a pattern P, as follows:

$$
\begin{array}{l}
\mathsf{slice} : P \to \{\texttt{-},\texttt{+}\} \to \mathit{token\ list} \\
\mathsf{token} : \mathit{token} \to A \to \{\texttt{-},\texttt{+}\} \to \mathit{token}
\end{array}
$$

$$
\begin{array}{lcl}
\mathsf{slice}(exp_1^A\ \texttt{(}^A\ exp_2^A\ \texttt{)}^A, op) & = & \mathsf{token}(exp_1, A, op)\ \mathsf{token}(\texttt{(}_1, A, op) \\
 & & \mathsf{token}(exp_2, A, op)\ \mathsf{token}(\texttt{)}_1, A, op)
\end{array}
$$

$$
\begin{array}{lcllcl}
\mathsf{token}(t, \texttt{-}, \texttt{-}) & = & t & \mathsf{token}(t, \texttt{-}, \texttt{+}) & = & \varepsilon \\
\mathsf{token}(t, \texttt{+}, \texttt{-}) & = & \varepsilon & \mathsf{token}(t, \texttt{+}, \texttt{+}) & = & t \\
\mathsf{token}(t, \texttt{-+}t'*, \texttt{-}) & = & t & \mathsf{token}(t, \texttt{-+}t'*, \texttt{+}) & = & t'* \\
\mathsf{token}(t, \varepsilon, \texttt{-}) & = & t & \mathsf{token}(t, \varepsilon, \texttt{+}) & = & t
\end{array}
$$

When taking the minus (`-`) slice, we obtain the tokens with a `-`, `-+`, or no annotation, and when taking the plus (`+`) slice, we obtain the sequence of tokens with a `+` annotation or no annotation, as well as the tokens in a `-+` annotation. A valid slice is one that is an element of the language generated by the grammar G: $G ::= exp\ \texttt{(}\ exp\ \texttt{)}$, $exp ::= \mathsf{id} \mid \mathsf{metaid}$.

```
@double_enable1 exists@
expression e,e1,e2,e3,e4;
position p;
position m1,m2,m3,m4,m5,m6,p1,p2;
@@
  e =
- pci_enable_msix@m1@p(e1, e2, e4)@m2
+ pci_enable_msix_range@p1(e1, e2, e3, e4)@p2
  ...
- e4@m3 = e@m4
  ...
- pci_enable_msix@m5(e1, e2, e4)@m6
```

Figure 9: Inserted position variables in `double_enable1`

5.3 Marking the start and end of each change

Our matching strategy uses Coccinelle to match the minus slice against the before code and the plus slice against the after code, independently. To synchronize the results for the two slices, with each other and with the changes indicated in the patch, we need to record the boundaries of the code annotated as `-` and `+` in the patch query. The Prequel compiler thus first introduces position variables to record these boundaries.

We refer to each sublist of successive tokens having a `-` or `+` annotation in the patch query as a *chunk*. The Prequel compiler separates each chunk into the sequence of constituent `-` tokens and the sequence of constituent `+` tokens. A pair of fresh position variables is generated for the start and end of each of these sublists, for each sublist that is not empty. The declaration of each such metavariable is associated with script code that checks whether the matched position is inside a hunk. This script code essentially makes Coccinelle, which sees only the before and after code individually, aware of the contents of the patch, and allows discarding immediately matches that are not relevant to the patch, and thus make the matching process more efficient.

The result of this phase, for each patch query rule, is a list of the introduced position variables, a record of which position variables represent the beginning and end of the `-` or `+` code in each hunk, and the rule with its updated pattern code. Figure 9 shows the result of adding position variables to the rule `double_enable1` of Figure 8.

5.4 Rule classification

The Prequel compiler next determines which rules of the patch query should be applied to the before code, to the after code, or to both. These classifications are organized into a lattice such that Unknown $\sqsubseteq$ Before, After and Before, After $\sqsubseteq$ Both. The choice among these classifications depends on multiple criteria: 1) the `-` and `+` annotations in the rule pattern code, 2) the before and after dependencies (*Dba*) indicated on the rule, and 3) the other rules that depend on the rule (*Dr*).

In the first step, an initial classification is chosen based on the pattern. If the pattern contains any `-+` tokens, or contains tokens annotated `-` as well as tokens annotated `+`, then the rule is classified as Both. If the pattern contains tokens annotated `-` (`+`, respectively), but no tokens annotated `+` (`-`, respectively) or `-+`, then the rule is classified as Before (After, respectively). Otherwise, the rule is annotated as Unknown.

In the second step, the initial classification is combined with explicit the Before-After dependencies (*Dba*). The explicit dependency, if any, has to be at least as general as the classification inferred from the pattern code. Thus, for example, a rule that contains only `-` and context code is incompatible with an annotation of After, which would imply that the rule should only be

applied to after code. On the other hand, a rule that contains only - and context code can be annotated as Both, implying that the - and context code will be matched against the before code, and the context code will be matched against the after code. Such a rule is subsequently considered to be Both.

The final step applies only to rules for which the classification is still Unknown. For such rules, we further take into account the classifications of any rules that depend on (Dr) the unknown rule. Working from the last rule back to the first, the following judgments collect an environment of the classification constraints induced by Before, After, and Both rules on the rules that they depend on. The final mapping of rules to classifications is ρ_r, defined by $\rho_0 \vdash SP : \rho_r, \rho_d$, where ρ_0 is an environment mapping the name of each rule to its classification based on the first two steps, and where ρ_d returns Unknown for any rule name for which it has no information. The relation $\vdash$ is defined as follows:

$$\frac{\rho_r(rule) = \text{Unknown} \qquad \rho_r \vdash SP : \rho'_r, \rho_d}{\begin{array}{l}\rho_r \vdash (rule \times Dba \times Dr \times M \times P)\ SP : \\ \quad \rho'_r[rule \mapsto \rho_d(rule)], \rho_d[r \mapsto \rho_d(r) \sqcup \rho_d(rule) \mid r \in Dr]\end{array}}$$

$$\frac{\rho_r(rule) \neq \text{Unknown} \qquad \rho_r \vdash SP : \rho'_r, \rho_d}{\begin{array}{l}\rho_r \vdash (rule \times Dba \times Dr \times M \times P)\ SP : \\ \quad \rho'_r, \rho_d[r \mapsto \rho_d(r) \sqcup \rho'_r(rule) \mid r \in Dr]\end{array}} \qquad \rho_r \vdash \varepsilon : \rho_r, \emptyset$$

In the full Prequel language, inherited metavariables provide an additional source of information for the classification of Unknown rules, analogous to dependencies. Unlike the case of dependencies, an inherited metavariable used only in the minus (plus, respectively) slice of a Both rule only provides a Before (After, respectively) classification requirement to the Unknown rule. An inherited metavariable thus only entails a Both classification requirement if it is used in both the minus slice and the plus slice of the Both rule.

Once the rules have been classified, this step ends with a check that every Before rule has a valid minus slice (see Section 5.2), every After rule has a valid plus slice, and every Both rule has both a valid minus splice and a valid plus slice. A warning is given if there are any Unknown rules, and such rules are not included in the generated Coccinelle semantic patch.

In our example in Figure 8, all rules can be classified immediately based on their pattern code. `Double_enable1`, `double_enable2`, and `unanticipated` are classified as Both, and `should_be_exact` is classified as After.

5.5 Rule splitting

The rule splitting step creates the SmPL rules that will be matched individually against the before and after code. For rules annotated Before (respectively, After), the rule splitting step simply generates the minus (respectively, plus) slice of the rule, which must be valid, and adds the before (respectively, after) dependency.

The treatment of a Both rule is more complex, because the single rule is split into two. The first step is to extract the minus and plus slices, as described in Section 5.2, both of which must be valid. As every SmPL rule must have a unique name, these are then renamed, by prepending `before_` and `after_`, respectively, to the original name. The `before_` rule is placed in the generated semantic patch before the `after_` rule. Because both need to match, the `after_` rule is updated to additionally depend on the `before_` one. Furthermore, any `fixed` metavariable that is used in both the minus and plus slices is declared in the `before_` rule and inherited in the `after_` one. Otherwise, the metavariable list is copied into the before and after rules, and specialized to the metavariables that the slice actually uses. Finally, later rules that depend on the Both rule have their dependencies updated to depend on either the `before_` or `after_` rule, according to the classification of the later rule.

```
@before_double_enable1 depends on before exists@
expression e,e1,e2,e4;
position p;
position m1,m2,m3,m4,m5,m6;
@@
  e = pci_enable_msix@m1@p(e1, e2, e4)@m2
  ...
  e4@m3 = e@m4
  ...
  pci_enable_msix@m5(e1, e2, e4)@m6

@after_double_enable1 depends on after && before_double_enable1 exists@
expression before_double_enable1.e,before_double_enable1.e1,before_double_enable1.e2,e3,before_double_enable1.e4
    ;
position p1,p2;
@@
  e = pci_enable_msix_range@p1(e1, e2, e3, e4)@p2

[...]

@before_unanticipated depends on before@
expression e,e1,e2,e4;
position p != {before_double_enable1.p,before_double_enable2.p};
position m1,m2;
@@
  e = pci_enable_msix@m1@p(e1, e2, e4)@m2

@after_unanticipated depends on after && before_unanticipated@
expression before_unanticipated.e,before_unanticipated.e1,before_unanticipated.e2,e3,before_unanticipated.e4;
position q != should_be_exact.q;
position p1,p2;
@@
  e = pci_enable_msix_range@p1@q(e1, e2, e3, e4)@p2
```

Figure 10: Result of splitting the patch query of Fig. 8. Position metavariable constraints are omitted for conciseness.

Figure 10 shows the result of applying these transformations to the patch query of Figure 8, focusing on the rules derived from `double_enable1` and `unanticipated`.

5.6 Consistency checking and output generation

Finally, several kinds of script rules are generated for each patch query rule to ensure the consistency between the matches of the before and after code.

The first set of script rules check that each `after_` rule matches when the corresponding `before_` one has matched, overall, and for each possible binding of the `fixed` metavariables. For each fixed metavariable, a rule is generated that discards the entire match instance if the corresponding `after_` rule does not match for the given fixed metavariable value.

The second generated script, for a given patch query, checks that the code annotated `-`, `+`, or `-+` in the original patch query is matched against code in lines that are changed by the commit. At run time, the generated semantic patch is provided with information about the start and end lines of each hunk, which it stores in a table during initialization. For each pair of starting and ending positions derived from a chunk that contains only `-` tokens or only `+` tokens (see Section 5.3), the script checks that both the starting position and the ending position are in the same hunk. For each pair of pairs of starting and ending positions derived from a chunk that contains both `-` and `+` tokens, the script checks that both the starting position and the ending position for each pair are in the same hunk, and that the positions from both pairs are in the same hunk, thus making the connection between the before and after code. If any of these checks fail, the entire match fails. If all of them succeed, the indices of the matched hunks are stored in a variable that

is exported by the script rule.

As noted in Section 5.1, PQL language, like SmPL, contains disjunction patterns, which can match any of a set of patterns and which contain - or + annotations. In this case, some parts of the pattern containing transformations might not be matched, raising the possibility that the script will inherit metavariables that are not bound. Originally, a Coccinelle script rule would only be applied if all of the inherited metavariables had values. To allow some metavariables to be unbound, without having to explicitly consider all of the permutations of bound and unbound variables, we have added to Coccinelle the ability to specify default values for script metavariables.

The full SmPL language also provides path operators, such as "...", which, when all paths are considered, can result in position variables being matched at multiple places in the code and thus having multiple values. Because positions are added to the patch query on contiguous sequences of tokens, they match unambiguously against the tokens in the before and after code, and values of the pairs of starting and ending position variables can also be aligned unambiguously, by sorting them by start and end line.

The last set of script rules emit the final result. A Prequel semantic patch may be composed of multiple rules that work together, such that a result is only desired if all of them match successfully. The Prequel compiler analyzes the dependencies between rules, and produces an output-generating script for rules on which no other rules depend. The script prints the list of matched hunks for all of the rules on which the selected PQL rule depends. The full PQL language supports negative dependencies, as illustrated by the rule `unanticipated` in Figure 8, which only depends on its predecessors by negative constraints on position variables. The hunks matched by these purely negative dependencies are not included in the output.

The scripts generated for the rule `unanticipated` from the patch query of Figure 8 are shown in Appendix A.

6 Execution of a Prequel Specification

The Prequel run-time system performs three main steps: 1) commit selection, 2) commit processing, and 3) result filtering. The main goals are to minimize the amount of code to which the patch query is actually applied, which is costly, and to filter the results to choose the most pertinent ones, according to criteria provided by the user. We describe these steps below.

6.1 Commit selection

Prequel first abstractly interprets the patch query to construct a formula describing tokens that must be present in the changed lines of code for the patch query to succeed on a given commit. For example, the formula associated with the patch query shown in Figure 8 is:

$$-\texttt{pci\_enable\_msix} \wedge +\texttt{pci\_enable\_msix\_range}$$

To ease subsequent processing, we represent the formula in conjunctive normal form (cnf). We omit further details about the abstraction process, due to space limitations.

The abstracted formula is then used in two steps: first with `git log -G` to do a coarse-grained selection of commits, and then by matching the formula against the selected patch code. Both steps are needed, because `git log -G` is fairly efficient, but accepts only one token (regular expression) and is not sensitive to whether the token is used in - or + code. On the other hand, our OCaml implementation of matching the formula against patches is slower, even when run in

parallel, but gives more precise results because it takes into account the whole formula and is sensitive to - and + code.

Because each instance of `git log -G` can search for uses of only one token, we must choose which token to use. For this, we adopt the heuristic that tokens that occur in fewer files in the current snapshot of the code base are likely also to be affected by fewer commits. Information about the number of files using a token in the current code snapshot can be collected quickly using the command `git grep -c`. For each conjunct in the cnf formula, we thus compute the sum of the number of files of each disjunct of the conjunct and then run `git log -G`, in parallel, for each disjunct of the conjunct with the lowest total file count. For example, with respect to the above formula, our Linux snapshot[5] contains 54 occurrences of `pci_enable_msix_range` and 8 occurrences of `pci_enable_msix`, and thus we run `git log -G` on `pci_enable_msix`. For almost all of examples considered in our evaluation (Section 7), only one token is chosen for use with `git log -G` by this process.

In some cases, such as when when the patch query contains only metavariables on the - and + lines, the abstract interpretation process does not produce any tokens for filtering the commits. Between Linux v3.0 and v4.4, for example, this amounts to almost 300,000 commits. Because it can take a long time for Coccinelle to process all of the affected files, Prequel can be parameterized with the maximum number of commits to consider.

6.2 Commit processing

For each selected commit, Prequel extracts the associated patch, the before and after version of each affected file, and a table matching each hunk in the patch to the name of the affected files and the line ranges affected by each addition or removal. Prequel then launches Coccinelle on each pair of before and after files, and the hunk information. Prequel then captures the standard output of Coccinelle, which consists of any print statements performed by the patch query's script code and a series of records reflecting the set of hunks matched by the patch query.

6.3 Result filtering

Beyond simply reporting the commits that match the patch query, Prequel has the goal of highlighting the matches that are most relevant to the user. To this end, Prequel can be parameterized by a threshold on the percentage of hunks that are matched, or a percentage of the changed lines that are in these hunks, and by whether this percentage should be computed from the number of hunks removing code, the number of hunks adding code, or the number of all the hunks in the patch. For example, for the patch query shown in Figure 5, we may prefer to compute the percentages in terms of only the hunks removing code, as we may want to see all of the variations, that replace calls to `pci_enable_msix`, no matter how complicated. On the other hand, a developer could instead prefer to see only the commits that make the fewest other changes overall, to start understanding the evolution of `pci_enable_msix` with the simplest cases.

As previously illustrated in Figure 6, when a commit is selected to be included in the output, Prequel also prints any standard output generated by the processing of the commit. Note that because of the result filtering phase, Prequel is not able to generate results incrementally. Rather it collects all of the results, and sorts them according to the provided criteria. This allows Prequel to show the most relevant commits among all of the matching ones.

[5] linux-next 20160318, commit id 5e3497c

7 Evaluation

We now evaluate Prequel, focusing on expressiveness and performance. Prequel is implemented in OCaml. The compiler is 2950 LOC, and relies on a modified version of the SmPL semantic patch compiler. The run-time system is 1004 LOC. All of our tests are run in parallel, one test per core, on a 48-core AMD Opteron(tm) Processor 6172, where each core has an 2.1 GHz CPU and each die has a 512 KB L2 cache. All experiments give the time for one run. Our experiments in this section consider all commits between Linux versions v3.0, released in July 2011, to v4.4, released in January 2016, covering 4.5 years and amounting to 280,901 commits. To illustrate the scope of Prequel, we consider three test suites: 1) a series of patch queries detecting removals of deprecated functions, 2) a series of patch queries derived from Coccinelle semantic patches written to perform automatic backporting of Linux device drivers, and 3) a series of patch queries based on the Coccinelle semantic patches found in the source code of the Linux kernel.

7.1 Analysis of deprecated functions

In Sections 3 and 4 we have compared the use of git and Prequel in understanding how to modernize the use of the deprecated function `pci_enable_msix`. We now carry out a larger scale analysis of deprecated functions, where we consider a deprecated function to be one that is used at least 10 times in at least one of Linux v3.0, v3.5, v3.10, v3.15, or v4.0, and where the number of uses in Linux v4.4 is less than one fifth of the largest number of uses in one of these previous versions. 95 functions, listed in Appendix B, satisfy these criteria.

For understanding the evolution of the deprecated functions, we consider two strategies, one that follows the pattern of the patch query in Figure 5, which checks that the return value is stored in an identical assignment in the before and after code (return value strategy), and a more general, but less precise, strategy that checks that the deprecated function and the function that replaces it have at least one argument in common (argument strategy). We configure Prequel such that there is no minimum match rate, and the match rate is computed as the percentage of hunks that remove code that are matched.

Figure 11 shows the highest match rate achieved by any commit between Linux v3.0 and v4.4, *i.e.* the match rate of the first result returned by Prequel. The x-axis in this graph and all subsequent ones represents the deprecated functions, in the order in which they appear in Appendix B. We have sorted the functions in the order of the higest match rate obtained with the argument strategy. For 8 functions, we obtain a patch with a 100% match of the return value patch query, and for 15 functions, we obtain a patch with a 100% match of the less precise argument patch query. Furthermore, for 12 functions the return value patch query identifies multiple replacement functions for the deprecated function; for the argument patch query, 34 of the deprecated functions have this property. This information can make the developer immediately aware of where there are choices to make in the evolution process. For 35 of the deprecated functions, we obtain no result at all. One reason is that the deprecated function was only used in files that were removed completely.

Finally, for each deprecated function, we have used `git log -G` to find patches that remove calls to the function. To reduce the running time, we have limited the number of commits returned by `git log -G` to 100. Figure 12 shows the number of patches that have to be skipped before finding the one that affects the minimum percentage of other lines of code. While 10 or fewer patches need to be skipped in many cases, 91 need to be skipped in the worst case. Furthermore, the user does not know in advance which are the most concise patches. The match rates provided by Prequel give an overview of the closest matching commits at once, while a git user may need to scan beyond the closest matching one, in case better matches occur later in

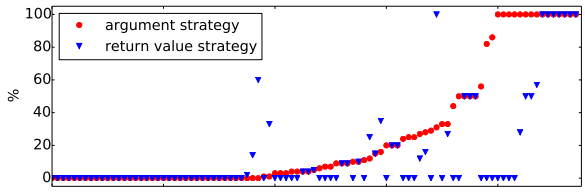


Figure 11: Highest match rate of the patches identified by Prequel

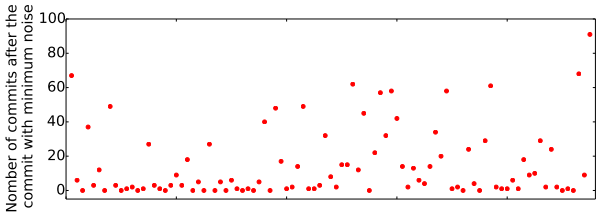


Figure 12: Number of commits that have to be skipped before reaching the closest match, out of the 100 most recent commits removing uses of deprecated functions

the commit history.

Performance. The main costs of Prequel are 1) the cost of `git log -G` in commit selection (Section 6.1), 2) the cost of commit filtering according to the abstracted formula, and 3) the cost of running Coccinelle on the compiled patch query and the files affected by the selected commits.

The cost of `git log -G` appears to be independent of the string being searched for. The average run time for a single invocation of `git log -G`, is 530 seconds with standard deviation 22, over the queries performed in the experiments described in Section 7.3.[6]

The cost of the second filtering step is at most 14.2 seconds for the deprecated functions, and is typically well below 1 second. In the case of deprecated functions, the formula inferred in the commit selection step always contains only one string, *i.e.*, the name of the deprecated function, which is annotated with `-`, as it appears in the patch query. We also give the name of this function to `git log -G`, so the only improvement produced by the second filtering step is to remove the commits in which a reference to the function is added rather than removed. Figure 13 contains a bar for each deprecated function in which the top of the bar represents the number of commits identified by `git log -G` and the bottom of the bar represents the number of commits remaining after the second filtering step. Up to 76 commits are removed from consideration by the second filtering step. This reduces the amount of code that has to be processed by Coccinelle, and thus reduces the Coccinelle execution time.

The cost of running Coccinelle on the compiled patch query for the different deprecated functions is shown in Figure 14. All of the patch queries for the different deprecated functions have the same structure, so the Coccinelle running time depends mainly on the number of relevant commits and the sizes of the affected files.

Excluding the `git log -G` time, the worst case execution time of the most costly operations

[6] In two cases, we observed roughly double this time, because Prequel makes two queries in parallel, but the use of `make -j` limits both forked processes to the same core.

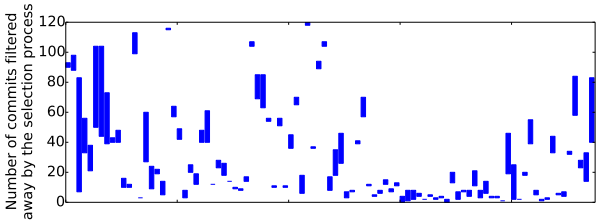


Figure 13: Commits after the first (top of bar) and second (bottom of bar) steps of the commit selection process

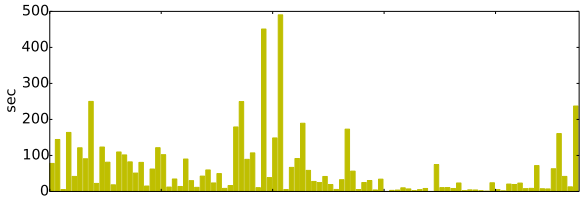


Figure 14: Running time of Coccinelle on the compiled patch query code

of Prequel for the deprecated functions is 581 seconds, *i.e.*, under 10 minutes. We plan to investigate whether indexing could replace `git log -G` and reduce the initial commit selection time. While 10 minutes is a bit long for interactive use, the user can do other things during this time, and is spared the burden of scrolling through many irrelevant commits.

7.2 Backports

Backporting is the process of porting modern code to an older kernel, for use in environments where stability requirements prevent a kernel upgrade. The Linux kernel backports project provides patches for backporting a range of device drivers from the latest kernel version to all Linux kernel releases since v3.0.[7] Currently, 54% of the changes needed for backporting are implemented using Coccinelle semantic patches [15]. None of the semantic patches used in this experiment were written by the developers of either Prequel or Coccinelle, but rather by professional Linux developers, showing the ability of professional Linux developers to effectively use the notation chosen by Prequel. We cannot expect to find the exact changes described by the backports project semantic patches in the Linux kernel, because the backports project targets its own library, that essentially sandboxes code that is not compatible with earlier kernels. Nevertheless, the code that a backport needs to remove is code that was added at some point in the evolution of the Linux kernel, and we can use Prequel to find examples of how that addition was accounted for in contemporaneous in-kernel code. Concretely, we transform the 27 backports semantic patches such that the minus slice becomes the plus slice, and drop the original plus slice completely. These semantic patches are listed in Appendix C.

Prequel is not able to infer any keywords or any sufficiently rare keywords for use in the commit selection process for three of the backports patch queries, resulting in tens or hundreds of thousands of commits to consider using Coccinelle; we aborted the evaluation in these cases.

[7] https://git.kernel.org/pub/scm/linux/kernel/git/backports/backports.git, commit id a91a3e6

Two other patch queries revealed bugs in the Prequel compiler, which we are investigating. For the remaining 21 patch queries, the Coccinelle running time ranges from 6 seconds to 3444 seconds, with the average, excluding the outlier of 3444 seconds, being 48 seconds. Relevant commits are found in 18 cases, with 100% matches when considering all hunks found in three cases.

As the backports patch queries are more complex than the ones used in studying the deprecated functions, the abstracted formulas constructed in the commit selection process contain multiple choices for the initial filtering with `git log -G`. As described in Section 6.1, we use the number of occurrences of a string in the current snapshot of the code base as a heuristic to estimate how many commits are relevant to the string in the commit history. To assess this heuristic, we have selected at random 1000 function names, 1000 structure field names, and 1000 #define constant names from the entire Linux kernel source code, and computed the correlation between the number of occurrences and the number of relevant commits. For functions, we obtain a correlation coefficient of 0.7390, which suggests some correlation, and for fields and constants, we obtain the lower values of 0.5503 and 0.6276, respectively. For the backports patch queries specifically, we obtain from `git log -G` at most 55386, 235, and 226 commits, and otherwise never more than 83. The heuristic thus seems to be sufficient in practice.

7.3 Semantic patches from the Linux kernel

To assess the general expressiveness of Prequel as well as the robustness of the Prequel compiler, we test it on the set of semantic patches found in the Linux kernel (listed in Appendix D). Unlike the backports case, we use the semantic patches as patch queries unchanged. These semantic patches were developed prior to and independently of Prequel. This experiment thus measures whether Prequel is able to handle a wide variety of specifications, written by a variety of developers, and the degree to which knowledge of how to use Coccinelle is transferable to Prequel.

Linux 4.4 includes 56 Coccinelle semantic patches, of which 34 provide a *patch* mode that performs a transformation. Of these, 10 fail due to the constraints on disjunctions noted in Section 5.1, one fails due to the inability to classify a rule as Before, After, or Both (Section 5.4), and Prequel aborts after the filtering step on four because the formula abstracted in the commit selection step (Section 6.1) did not enable `git log -G` to do enough filtering, resulting in over one hundred thousand commits to consider using Coccinelle. For the remaining 17 patch queries on which Coccinelle was invoked, the Coccinelle running time ranges from 0 seconds to 4978 seconds, with the average being 441 seconds and the median being 82 seconds. The average memory usage, computed using /usr/bin/time -v, is 9677 bytes, with standard deviation 123, for semantic patches that do not satisfy the constraints of PQL, 89,686, with standard deviation 140, for semantic patches that Prequel rejects due to insufficient filtering, and 1,278,692 bytes, with standard deviation 1997, for semantic patches on which Prequel completes normally. Relevant commits are found in 14 cases, with 100% matches when considering all hunks found in twelve cases.

8 Related Work

The Google code review tool Gerrit provides a language for querying a change database [5]. Queries describe change metadata, such as age and reviewing status (open, reviewed, etc). Prequel, in contrast, focuses on patterns in source code. It could be useful to combine the two approaches, to allow Gerrit to additionally reason about the changes themselves.

Uquillas Gómez *et al.* propose Time Warp for reasoning about code histories. Their approach layers a metamodel representing the different versions of the code over time over a metamodel representing various aspects of the code structure. This approach makes it possible to reason about relationships over time in the code history, such as the existence of a method that is called in an early version, not called in later versions, and then becomes called again, possibly indicating the introduction of the use of a deprecated function that should be removed. The underlying model of the code, however, exposes only high level properties such as inheritance relationships and the existence of method calls and attribute references, implying that it is not possible to reason about control and dataflow relationships within the changed code, as is enabled by Prequel.

Sadowski *et al.* have used surveys and logs of developers at Google to study how developers search for code. They found that developers search frequently, on average 12 times per day, a result corroborated by a previous study [16], that 34% of the time they are searching for coding examples, which is greater than any other motivation for search, that developers often know where to find the information they need, as over 26% of search queries mention a specific file, and that search queries are constructed incrementally, as more information is accumulated. While the study of Sadowski *et al.* focused on searches in code snapshots and code metadata, these trends may also apply to searching within patches. In particular, if a developer can restrict a search to a particular directory or particular file, then that will reduce the running time of Prequel. Furthermore, the fact that PQL looks like C source code should make it easy to refine a query as more information becomes available.

A number of approaches have considered how to identify occurrences of refactorings in the history of a code base [6, 14, 17, 18]. A refactoring is a semantics preserving code change that has the goal of improving the program structure. 72 common refactorings have been identified and codified by Fowler [4]. Refactoring detection thus requires analysis of changes in a code history. The most comprehensive refactoring detection tool, in terms of the number of refactorings covered, is Ref-Finder, of Prete *et al.* [14], which can identify instances of 63 of Fowler's refactorings, with high precision and recall. Ref-Finder relies on a database of facts about the software, and the encoding of the characteristics of a refactoring as a logic-based query over that database. Unlike Prequel, Ref-Finder is not intended to be programmable by the user. Thus, the facts and predicates only express properties that are specific to refactoring detection. Ref-Finder has also only been used to search for all refactorings in a pair of versions, that appear to be released fairly close in time. Most experiments take less than a minute, but one takes more than an hour. The approach may thus not scale up to a history of 4.5 years as we have considered in the evaluation of Prequel. Another line of work has focused on the problem of API migration [11, 19]. These approaches infer properties from changes in control-flow graphs, and thus intrinsically only detect changes in the set of names of relevant function calls. These approaches thus do not address the problem of how to adjust other required computations, such as the computation of function arguments.

Padioleau identified the problem of collateral evolutions [13] in Linux device drivers. Such evolutions are changes needed in response to changes in the interface of library functions. The study of collateral evolutions in Linux device drivers then motivated the development of Coccinelle as a means of expressing the changes required [12]. The effort on Coccinelle, however, did not provide a means of finding examples of how to carry out these changes. This is the functionality that is offered by Prequel.

Prequel relies on the line-based `diff` algorithm provided by git. The information provided by this algorithm is approximate, because often some parts of a line actually do not contain changes. Tree-based difference algorithms, such as GumTree [3], provide differencing information at the level of language constructs, such as statements and expressions. Tree-based differencing has

been found to be much more expensive than line-based `diff`, due to the need to parse the code. Furthermore, when the user does not have exact knowledge of what he is searching for, he may end up marking as - or + terms that are adjacent to changes, but that do not actually change themselves. For example, in the patch query in Figure 8, complete function calls are annotated as - or +, even though several of the arguments do not change before the before and after code.

9 Conclusion

In conclusion, we have presented a patch query language PQL and the associated runtime system Prequel for searching in commit histories. The main contributions of Prequel are to be able to take into account context information and to be able to control the rate of unmatched changes in the reported commits. We have illustrated how Prequel could be used to explore how to modernize uses of a particular API function. We have also shown that Prequel is reasonably efficient, although optimization opportunities, such as patch indexing have not yet been fully explored.

While we shown that Prequel is useful on the problem of modernizing out of date code, we expect that it can be applied in other situations that require history information, such as identifying potential software development trouble spots based on characterization of recent bug-fix patches. We will explore such applications in the future, as well as exploring strategies for improving the overall performance.

Acknowledgments

This work was supported in part by the Open Source Automation Development Lab (OSADL), in the context of the project SIL2LinuxMP. We thank Tegawendé F. Bissyandé for his feedback on a draft of this document.

References

[1] J. Brunel, D. Doligez, R. R. Hansen, J. L. Lawall, and G. Muller. A foundation for flow-based program matching: using temporal logic and model checking. In *POPL*, pages 114–126, 2009.

[2] D. R. Engler, D. Y. Chen, and A. Chou. Bugs as inconsistent behavior: A general approach to inferring errors in systems code. In *SOSP*, pages 57–72, 2001.

[3] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus. Fine-grained and accurate source code differencing. In *ASE*, pages 313–324, 2014.

[4] M. Fowler. *Refactoring: Improving the Design of Existing Code.* Addison Wesley, 1999.

[5] Gerrit code review - searching changes, 2016. https://gerrit.googlecode.com/svn-history/r3021/documentation/2.1.4/cmd-query.html.

[6] S. Hayashi, Y. Tsuda, and M. Saeki. Detecting occurrences of refactoring with heuristic search. In *APSEC*, pages 453–460, 2008.

[7] How to get your change into the Linux kernel. https://www.-kernel.org/doc/Documentation/SubmittingPatches, Section 3.

[8] C. Le Goues and W. Weimer. Specification mining with few false positives. In *TACAS*, pages 292–306, 2009.

[9] Z. Li and Y. Zhou. PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code. In *ESEC/SIGSOFT FSE*, pages 306–315, 2005.

[10] D. MacKenzie, P. Eggert, and R. Stallman. *Comparing and Merging Files With Gnu Diff and Patch*. Network Theory Ltd, Jan. 2003.

[11] S. Meng, X. Wang, L. Zhang, and H. Mei. A history-based matching approach to identification of framework evolution. In *ICSE*, pages 353–363, 2012.

[12] Y. Padioleau, J. L. Lawall, R. R. Hansen, and G. Muller. Documenting and automating collateral evolutions in Linux device drivers. In *EuroSys*, pages 247–260, 2008.

[13] Y. Padioleau, J. L. Lawall, and G. Muller. Understanding collateral evolution in Linux device drivers. In *EuroSys*, pages 59–71, 2006.

[14] K. Prete, N. Rachatasumrit, N. Sudan, and M. Kim. Template-based reconstruction of complex refactorings. In *ICSM*, pages 1–10, Timisoara, Romania, 2010.

[15] L. R. Rodriguez and J. Lawall. Increasing automation in the backporting of Linux drivers using Coccinelle. In *11th European Dependable Computing Conference - Dependability in Practice (EDCC)*, 2015.

[16] J. Singer, T. Lethbridge, N. Vinson, and N. Anquetil. An examination of software engineering work practices. In *Proceedings of the 1997 Conference of the Centre for Advanced Studies on Collaborative Research, CASCON '97*, pages 21–37. IBM Press, 1997.

[17] K. Taneja, D. Dig, and T. Xie. Automated detection of API refactorings in libraries. In *ASE*, pages 377–380, 2007.

[18] P. Weissgerber and S. Diehl. Identifying refactorings from source-code changes. In *ASE*, pages 231–240, 2006.

[19] W. Wu, Y.-G. Guéhéneuc, G. Antoniol, and M. Kim. AURA: a hybrid approach to identify framework evolution. In *ICSE-Volume 1*, pages 325–334, 2010.

A Consistency checking and output generation scripts

For the rule `unanticipated` of Figure 8, Figure 15 shows the scripts generated to ensure the successful match of the `after` rule (lines 1-10), to check that the Prequel `-` and `+` lines match code that is in the same hunk (lines 12-28), and to generate the final output (lines 30-35).

B Deprecated functions

The deprecated functions considered are as follows, listed in the order in which they appear in all of the graphs of Section 7.1.

```
1 @script:ocaml check_before_unanticipated_after_unanticipated depends on before_unanticipated && !
      after_unanticipated@
2 @@
3 Coccilib.include_match false
4
5 @script:ocaml check_fixed_before_unanticipated_e depends on !after_unanticipated@
6 _e << before_unanticipated.e;
7 @@
8 Coccilib.include_match false
9
10 [...] // rules for each fixed metavariable
11
12 @script:ocaml check_after_unanticipated depends on after_unanticipated@
13 m1 << before_unanticipated.m1 = [];
14 m2 << before_unanticipated.m2 = [];
15 p1 << after_unanticipated.p1 = [];
16 p2 << after_unanticipated.p2 = [];
17 found_hunks;
18 @@
19 let res = ref [] in
20 let flag =
21   (let jm__0__1 = check_pair_in_same_hunk m1 m2 p1 p2 in
22   match m1 with
23       None -> false
24     | Some l -> (res := List.append l !res; true)) in
25 if not flag
26 then Coccilib.include_match false
27 else
28 found_hunks := make_ident (String.concat "," !res)
29
30 @script:ocaml commit_after_unanticipated@
31 hunks_for_after_unanticipated << check_after_unanticipated.found_hunks;
32 @@
33 start_record();
34 Printf.printf "%s\n" hunks_for_after_unanticipated;
35 end_record()
```

Figure 15: Consistency checking and output generation scripts for `unanticipated`

KERNEL_VERSION, clkdev_add_table, cpu_class_is_omap1, cpu_is_omap24xx, cpufreq_frequency_table_verify, ehci_port_power, fops_put, force_sigsegv, gameport_register_driver, gameport_unregister_driver, ip_set_timeout_uget, l2x0_of_init, lcd_device_unregister, mpc83xx_add_bridge, of_i2c_register_devices, of_irq_init, omap3_mux_init, omap3_pmic_get_config, omap3_pmic_init, omap_display_init, omap_hsmmc_init, omap_mux_init_gpio, omap_sdrc_init, phy_driver_unregister, phy_drivers_register, rdma_node_get_transport, s3c_sdhci2_set_platdata, serio_register_driver, serio_unregister_driver, timer_tick, to_mca_device, update_sched_clock, usb_bind_phy, usb_free_descriptors, usb_musb_init, RTA_DATA, phy_driver_register, get_clock, cpufreq_frequency_table_target, ehci_init, ip_set_optattr_netorder, regulator_bulk_free, v4l2_ctrl_query_fill, fsl_add_bridge, usb_copy_descriptors, nvkm_bios, nvkm_fb, nvkm_i2c, iio_device_free, regulator_unregister, sigorsets, rtc_device_unregister, __clk_get_flags, usb_register, mtrr_del, NLMSG_LENGTH, usb_deregister, au_readl, snd_soc_dai_set_fmt, iov_length, comedi_pci_disable, iio_device_alloc, usb_string_id, s3c24xx_init_clocks, dma_set_tx_state, videobuf_dma_unmap, virt_to_mfn, snd_soc_jack_add_pins, au_writel, irq_get_handler_data, init_page_count, NLMSG_SPACE, snd_soc_dapm_enable_pin, phy_drivers_unregister, mtrr_add, regulator_bulk_get, rtc_device_register, usbhid_submit_report, cpufreq_notify_transition, xfs_update_cksum, PDE, ata_dev_printk, comedi_event, gpio_line_set, tty_register_device, NLMSG_DATA, cpufreq_frequency_table_cpuinfo, pci_pcie_cap, PTR_RET, __constant_cpu_to_le16, __constant_cpu_to_le32, __constant_htons, devm_pinctrl_get_select_default, pci_enable_msix, regulator_register

Semantic patch	Notes
0001-group_attr_class	
0001-netlink-portid	
0002-gpio-parent	
0002-group_attr_bus	
0002-no_dmabuf	no filtering
0019-usb_driver_lpm	
0031-sk_data_ready	no filtering
0054-struct-proto_ops-sig	Prequel compiler error
0055-netdev-tstats	
0062-iff-no-queue	
0065-ndisc_send_na-argument	
0067-mdio-addr	
ethtool_cmd_mdix	
ethtool_eee	
features_check	
genl-const	
get_module	
get_ts_info	
igb_pci_error_handlers	no filtering
no-pfmemalloc	
ptp_getsettime64	
rxnfc	insufficient filtering
set_vf_rate.cocci	Prequel compiler error
set_vf_spoofchk.cocci	
skb_no_fcs.cocci	
skb_no_xmit_more.cocci	
sriov_configure.cocci	

Figure 16: Backports semantic patches

C Backports semantic patches

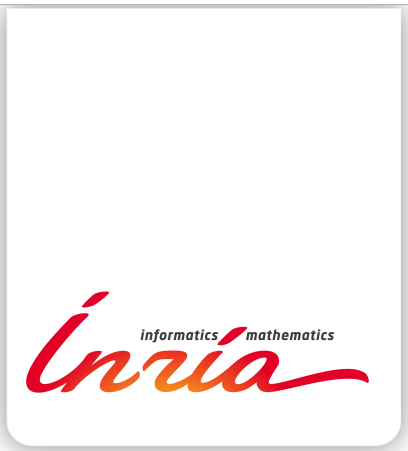
Figure 16 lists the backports semantic patches that are used in the experiments in Section 7.2.

D Linux kernel semantic patches

Figure 17 lists the semantic patches in the Linux kernel that are used in the experiments in Section 7.3. In each case, we give the number of lines of the semantic patch code used in the patch mode, excluding blank lines and comments.

Semantic patch	Lines of patch mode code	Notes
alloc_cast	6	insufficient filtering
array_size	17	disjunction error
badty	9	insufficient filtering
badzero	89	dead code
boolinit	58	disjunction error
boolreturn	20	disjunction error
bugon	5	
call_kern	55	
compare_const_fl	62	disjunction error
device_node_continue	52	
d_find_alias	43	
eno	8	
err_cast	5	
fen	44	
ifnullfree	19	
irqf_oneshot	52	disjunction error
itnull	42	disjunction error
kstrdup	24	
kzalloc-simple	10	
memdup	23	
memdup_user	21	
noderef	22	disjunction error
odd_ptr_err	40	disjunction error
of_table	20	disjunction error
platform_no_drv_owner	57	
pm_runtime	37	disjunction error
pool_zalloc-simple	18	
ptr_ret	12	
resource_size	5	
returnvar	12	no filtering
semicolon	50	no filtering
simple_open	24	
vma_pages	4	
warn	41	

Figure 17: Linux kernel semantic patches



RESEARCH CENTRE
PARIS – ROCQUENCOURT

Domaine de Voluceau, - Rocquencourt
B.P. 105 - 78153 Le Chesnay Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399